ROS-Industrial Advanced Developer's Training Class

August 2017

Southwest Research Institute

Session 5:
Advanced Topics
(Path Planning and Perception)

Southwest Research Institute

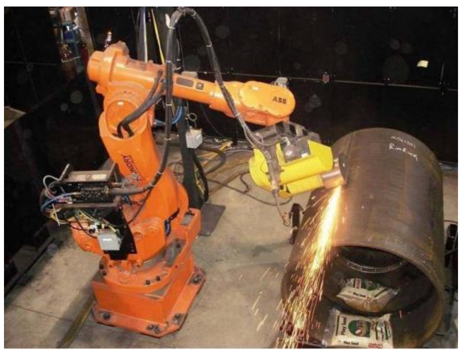
INTRODUCTION TO DESCARTES

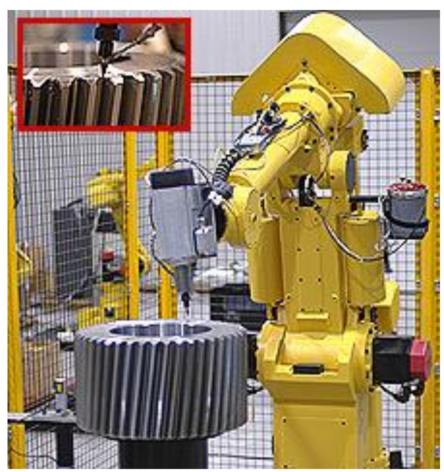
Outline

- Introduction
- Overview
  - Descartes architecture
- Descartes Path Planning
  - Exercise 5.0
- Perception
  - Exercise 5.1
- STOMP Path Planning
  - Exercise 5.2

Introduction

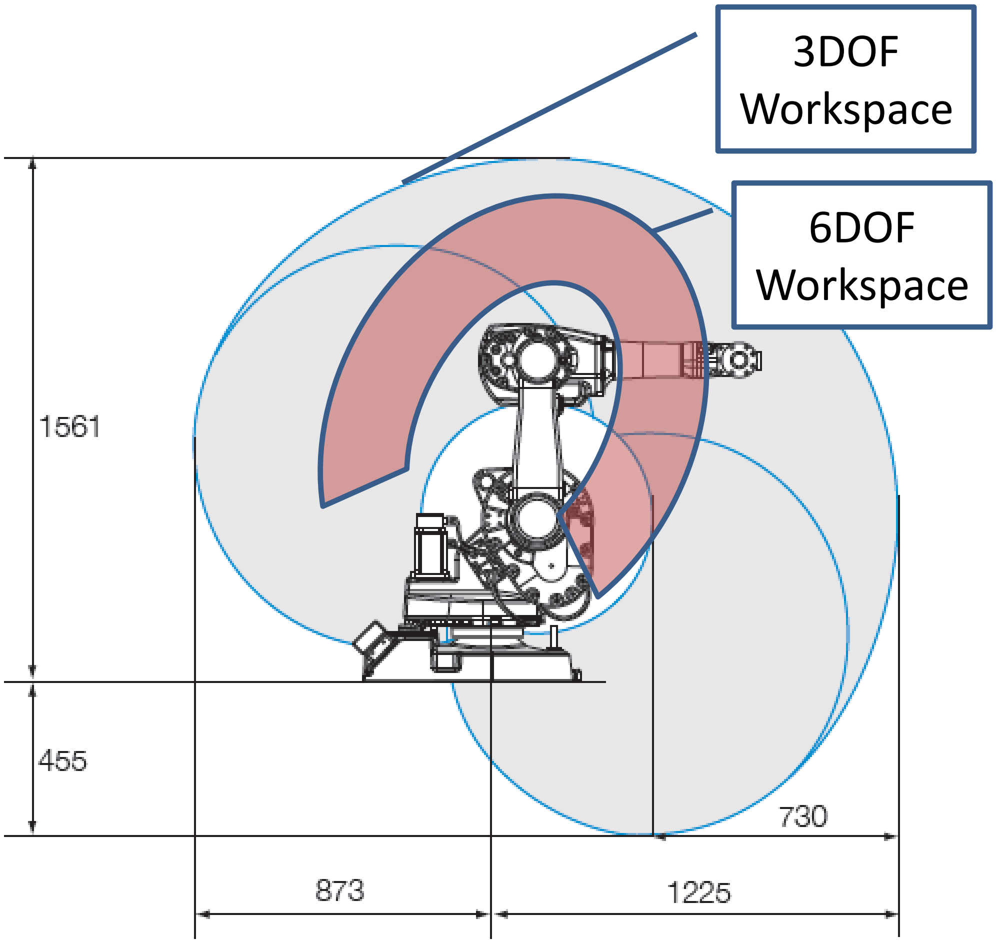
- Application Need:
  - Semi-constrained trajectories: traj. DOF < robot DOF





Current Solution

- Arbitrary assignment of 6DOF poses, redundant axes -> IK
- Limited guarantee on trajectory timing
- Limitations
  - Reduced workspace
  - Relies on human intuition
  - Collisions, singularities, joint limits

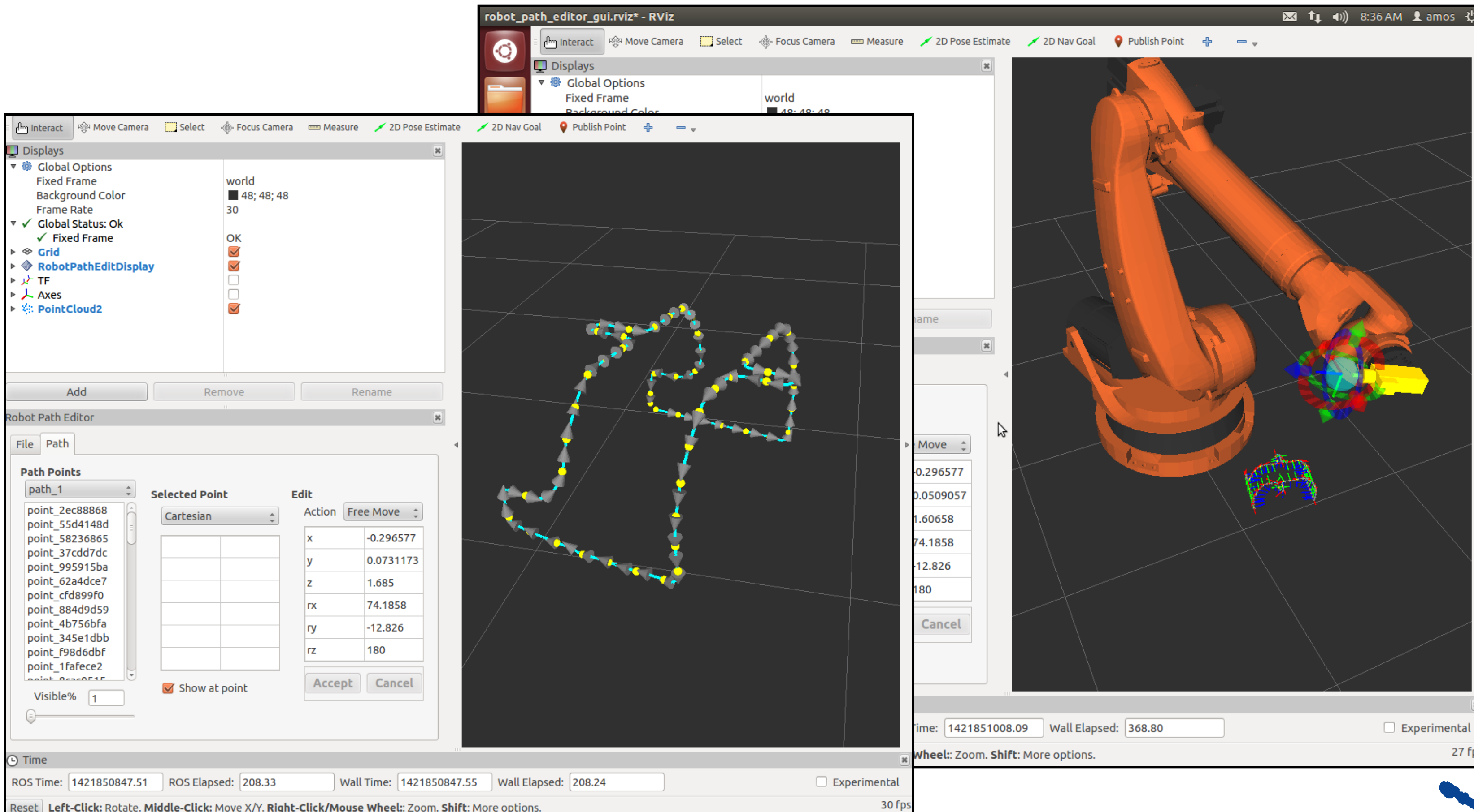


Descartes

- Planning library for semi-constrained trajectories
- Requirements
  - Generate well behaved plans that minimize joint motions
  - Find easy solutions fast, hard solutions with time
  - Handle hybrid trajectories (joint, Cartesian, specialized points)
  - Fast re-planning/cached planning

Descartes Use Case

- Robotic Routing



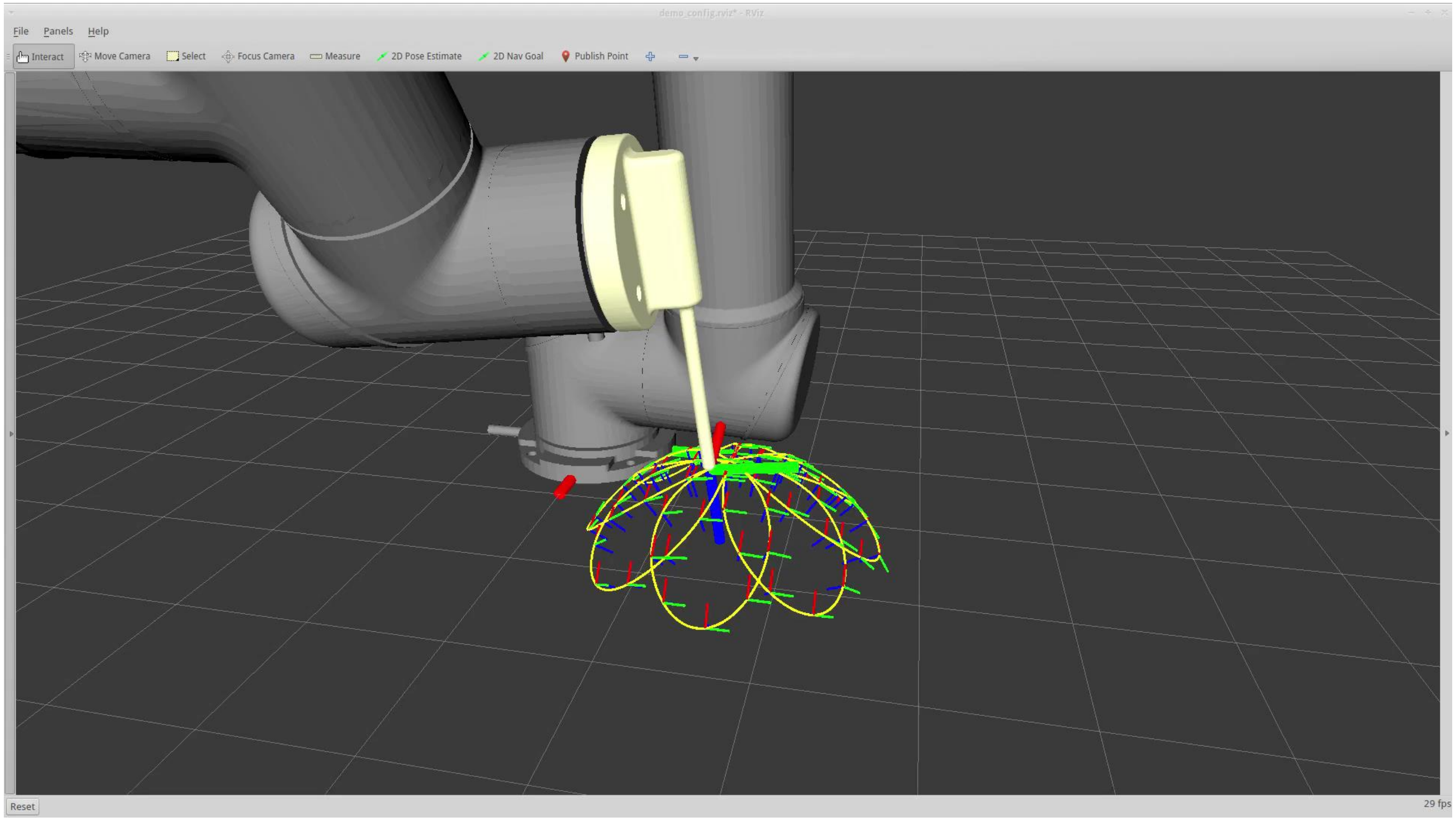
Other Uses

- Robotic Blending

Open Source Details

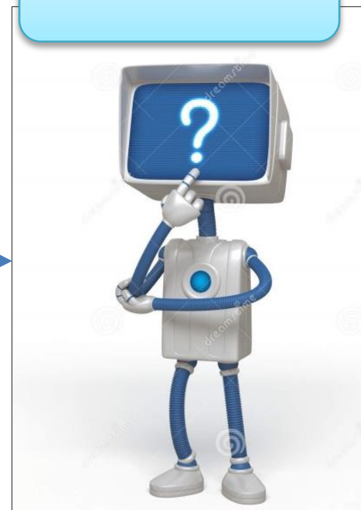
- Public development: https://github.com/ros-industrial-consortium/descartes
- Wiki Page: http://wiki.ros.org/descartes
- Acknowledgements:
  - Dev team: Dan Solomon (former SwRI), Shaun Edwards (former SwRI), Jorge Nicho (SwRI), Jonathan Meyer (SwRI), Purser Sturgeon(SwRI)
  - Supported by: NIST (70NANB14H226), ROS-Industrial Consortium FTP

Descartes Demonstration

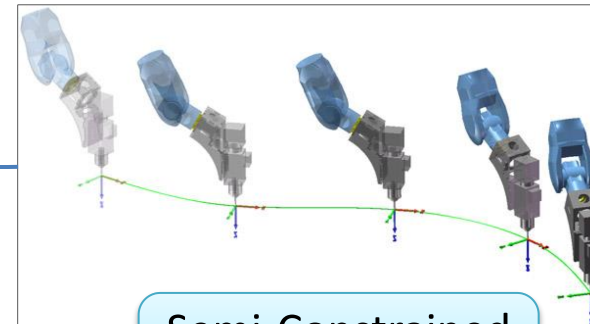


Descartes Architecture

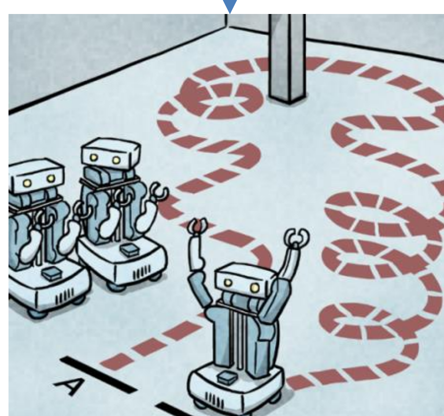
Path Planner



Robot Model



Semi-Constrained Trajectory



Robot Path

Descartes Interfaces

- Trajectory Point
  - Robot independent
  - Tolerance (fuzzy)
  - Timing
- Robot Model
  - IK/FK
  - Validity (Collision checking, limits)
  - Similar to MoveIt::RobotState, but with **getAllIK**
- Planner
  - Trajectory solving
  - Plan caching/re-planning

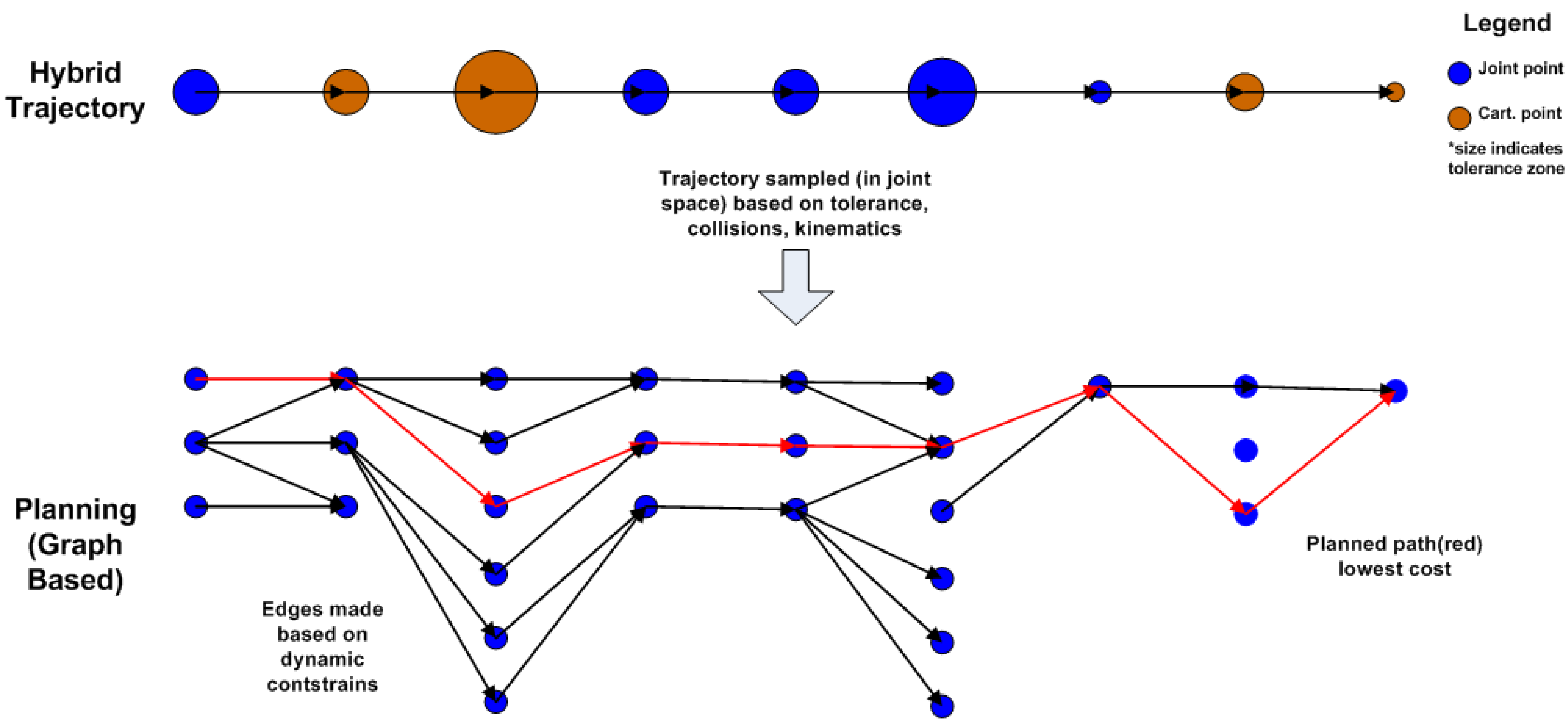
Descartes Implementations

- Trajectory Points
  - Cartesian point
  - Joint point
  - AxialSymmetric point (5DOF)
- Robot Model
  - MoveIt wrapper (working with MoveIt to make better)
  - FastIK wrappers
  - Custom solution
- Planners
  - Dense – graph based search
  - Sparse – hybrid graph based/interpolated search

Trajectory Point "Types"

- Trajectory Points
  - JointTrajectoryPt
    - Represents a robot joint pose. It can accept tolerances for each joint
  - CartTrajectoryPt
    - Defines the position and orientation of the tool relative to a world coordinate frame. It can also apply tolerances for the relevant variables that determine the tool pose.
  - AxialSymmetricPt
    - Extends the CartTrajectoryPt by specifying a free axis of rotation for the tool. Useful whenever the orientation about the tool's approach vector doesn't have to be defined.

Descartes Implementations



Cartesian Trajectory Point

- Create a **CartTrajectoryPt** from a tool pose.
- Store the **CartTrajectoryPt** in a **TrajectoryPtPtr** type.

```
descartes_core::TrajectoryPtPtr   cart_point_ptr ( new
        descartes_trajectory::CartTrajectoryPt ( tool_pose ));
```

Axial Symmetric Point

- Use the **AxialSymmetricPt** to create a tool point with rotational freedom about z.
- Use **tool_pose** to set the nominal tool position.

```
descartes_core::TrajectoryPtPtr  free_z_rot_pt(
    new descartes_trajectory::AxialSymmetricPt(
        tool_pose,
        0.5f,
        descartes_trajectory::AxialSymmetricPt::Z_AXIS));
```

Joint Point

- Use the **JointTrajectoryPt** to “fix” the robot’s position at any given point.
- Could be used to force a particular start or end configuration.

```
std::vector<double> joint_pose = {0, 0, 0, 0, 0, 0};
descartes_core::TrajectoryPtPtr   joint_pt(
    new descartes_trajectory::JointTrajectoryPt(joint_pose) );
```

Timing Constraints

- All trajectory points take an optional **TimingConstraint** that enables the planners to more optimally search the graph space.
- This defines the time, in seconds, to achieve this position from the previous point.

```
Descartes_core::TimingConstraint tm (1.0);
descartes_core::TrajectoryPtPtr   joint_pt(
    new descartes_trajectory::JointTrajectoryPt(joint_pose, tm) );
```

Robot Models

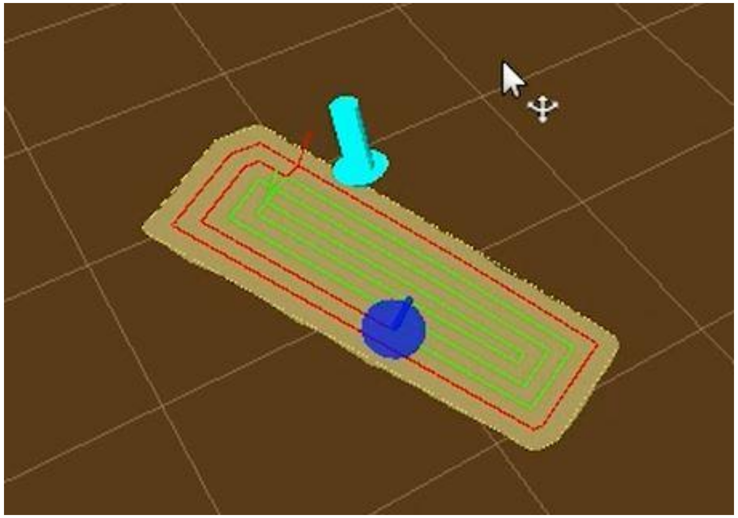
- Robot Model Implementations
  - **MoveitStateAdapter** :
    - Wraps moveit Robot State.
    - Can be used with most 6DOF robots.
    - Uses IK Numerical Solver.
  - **Custom Robot Model**
    - Specific to a particular robot (lab demo uses UR5 specific implementation).
    - Usually uses closed-form IK solution (a lot faster than numerical).
    - Planners solve a lot faster with a custom robot model.

Used in the Exercises

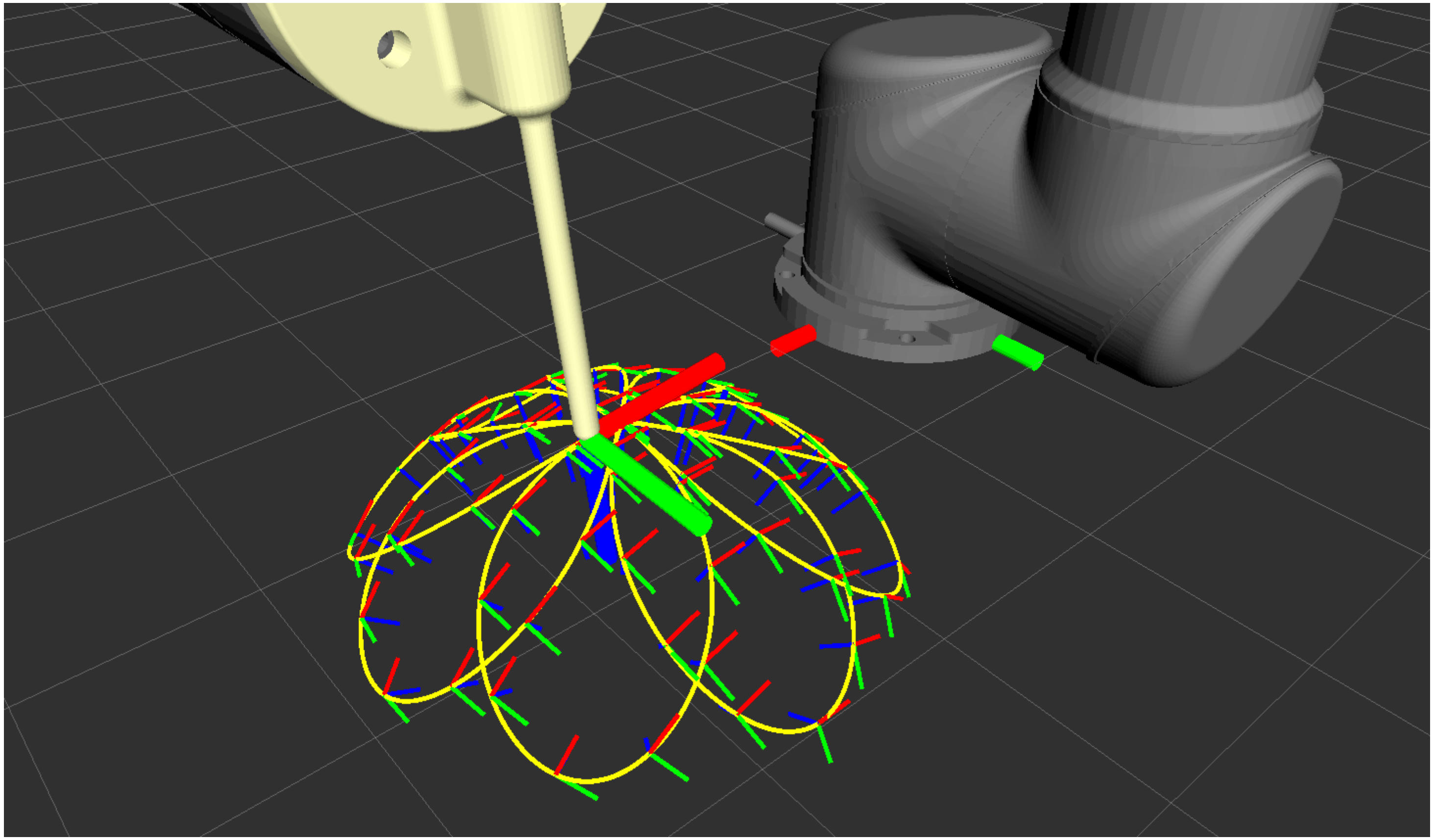
Used in the Lab

Descartes Input/Output

- Input:
  - Can come from CAD
  - From processed scan data
  - Elsewhere
- Output
  - Joint trajectories
  - Must convert to ROS format to work with other ROS components (see 4.0)



DESCARTES IMPLEMENTATIONS



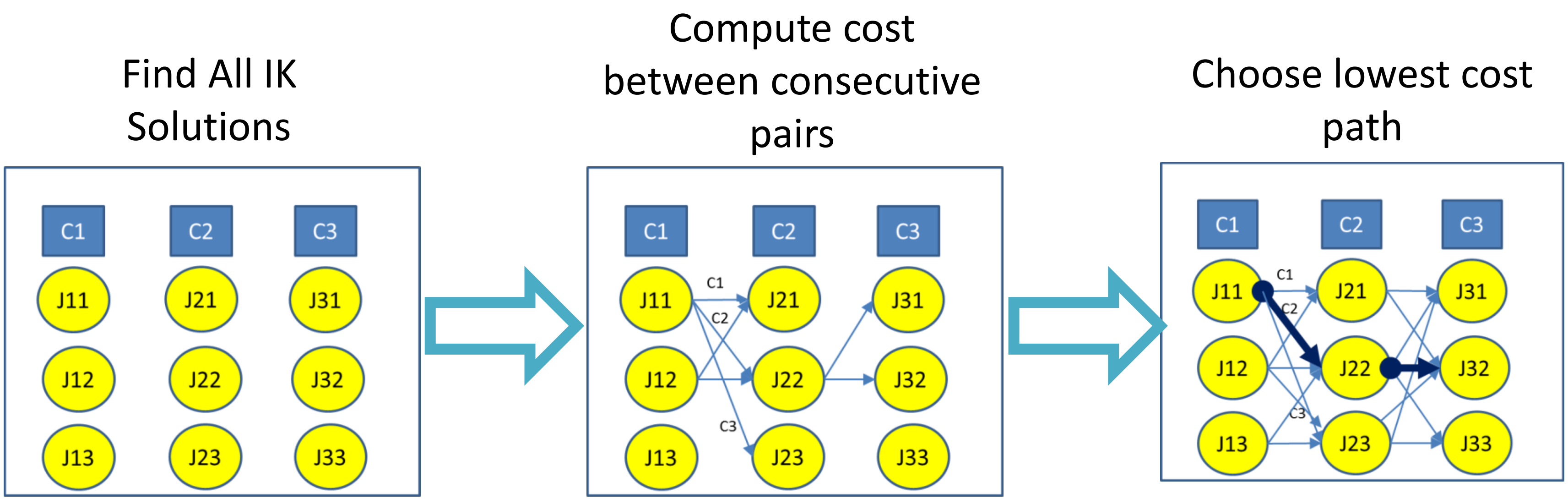
You specify these “points”, and Descartes finds shortest path through them.

Descartes Path Planning

- Planners
  - Planners are the highest level component of the Descartes architecture.
  - Take a trajectory of points and return a valid path expressed in joint positions for each point in the tool path.
  - Two implementations
    - DensePlanner
    - SparsePlanner

Descartes Path Planning

- Dense Planner
  - Finds a path through the points that minimizes the joint motion.



Descartes Path Planning

- Dense Planner
  - Search graph uses joint solutions as vertices and the movement costs as edges
  - Applies Dijkstra's algorithm to find the lowest cost path from a start to and end configuration.

Descartes Path Planning

- Create a trajectory of **AxialSymetricPt** points.
- Store all of the points in the **traj** array.

```
for( …)
 {
   …

   descartes_core::TrajectoryPtPtr  cart_point (
      new descartes_trajectory::AxialSymmetricPt (
                                        tool_pose ,
                                        1.0f,
                                        descartes_trajectory::AxialSymmetricPt::Z_AXIS));
   traj.push_back(cart_point);
 }
```

Descartes Path Planning

- Create and **initialize** a **DensePlanner**.
- Verify that initialization succeeded.

```
descartes_planner::DensePlanner planner;
 if (planner.initialize( robot_model_ptr ))
 {
              …
 }
```

Descartes Path Planning

- Use **planPath(…)** to plan a robot path.
- Invoke **getPath(…)** to get the robot **path** from the planner.

```
std::vector < descartes_core::TrajectoryPtPtr > path;
 if ( planner.planPath(  traj  ) )
 {
   if ( planner. getPath( path ) )
   {
               …
   }
               …
}
```

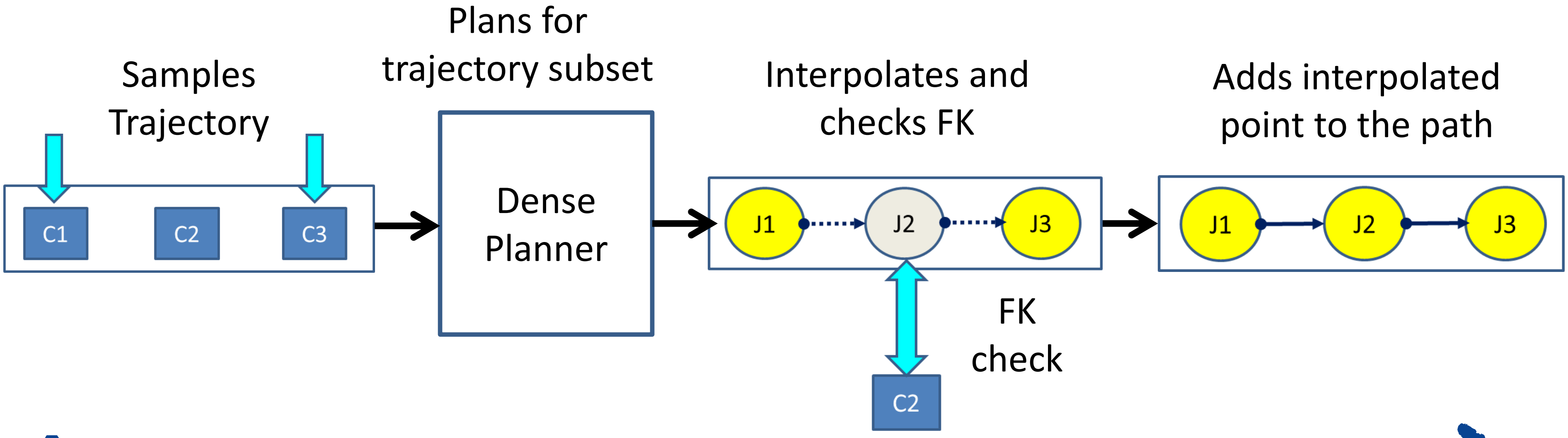
Descartes Path Planning

- Write a **for loop** to print all the joints poses in the planned path to the console.

```
std::vector< double >  seed ( robot_model_ptr->getDOF() );
for( … )
{
  std::vector <double> joints;
  descartes_core::TrajectoryPtPtr  joint_pt =  path[ i ];
  joint_pt -> getNominalJointPose (seed ,*robot_model_ptr , joints );

            // print joint values in joints
}
```

Descartes Path Planning

- Sparse Planner
  - Saves computational time by planning with a subset of the trajectory points and completing the path using joint interpolation.



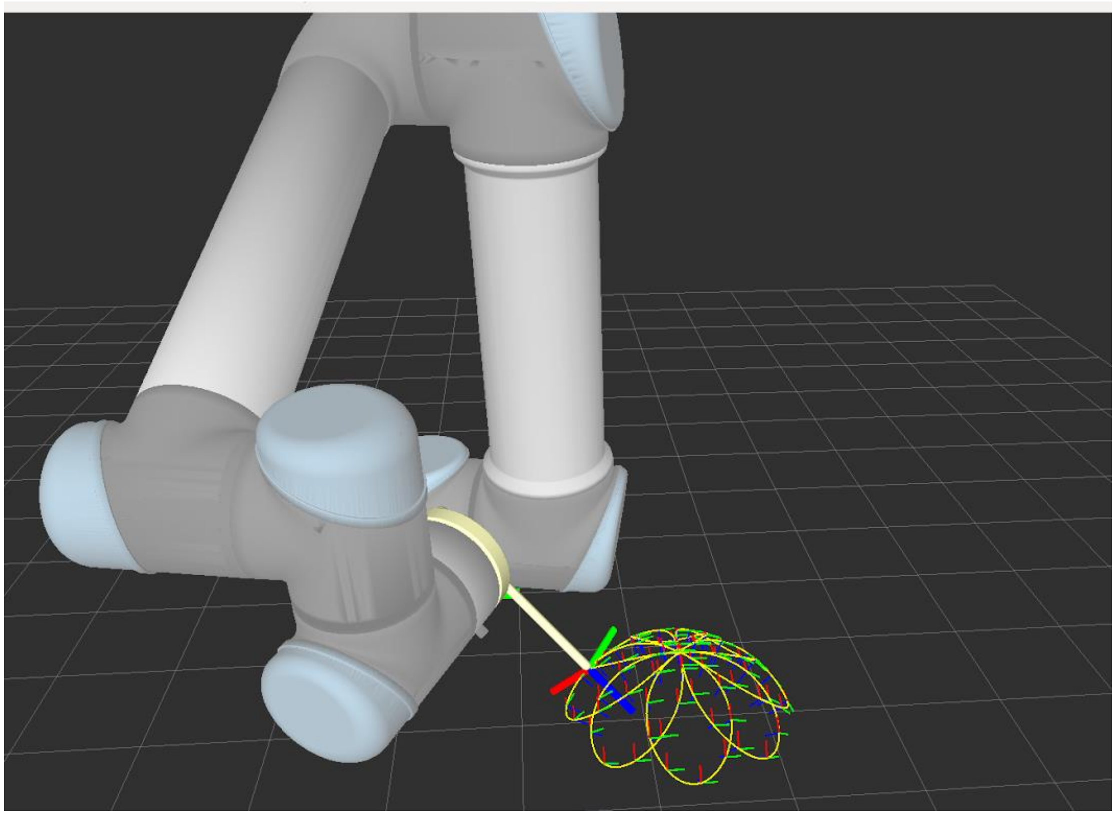
Exercise 5.0

- Go back to the line where the **DensePlanner** was created and replace it with the **SparsePlanner**.
- Planning should be a lot faster now.

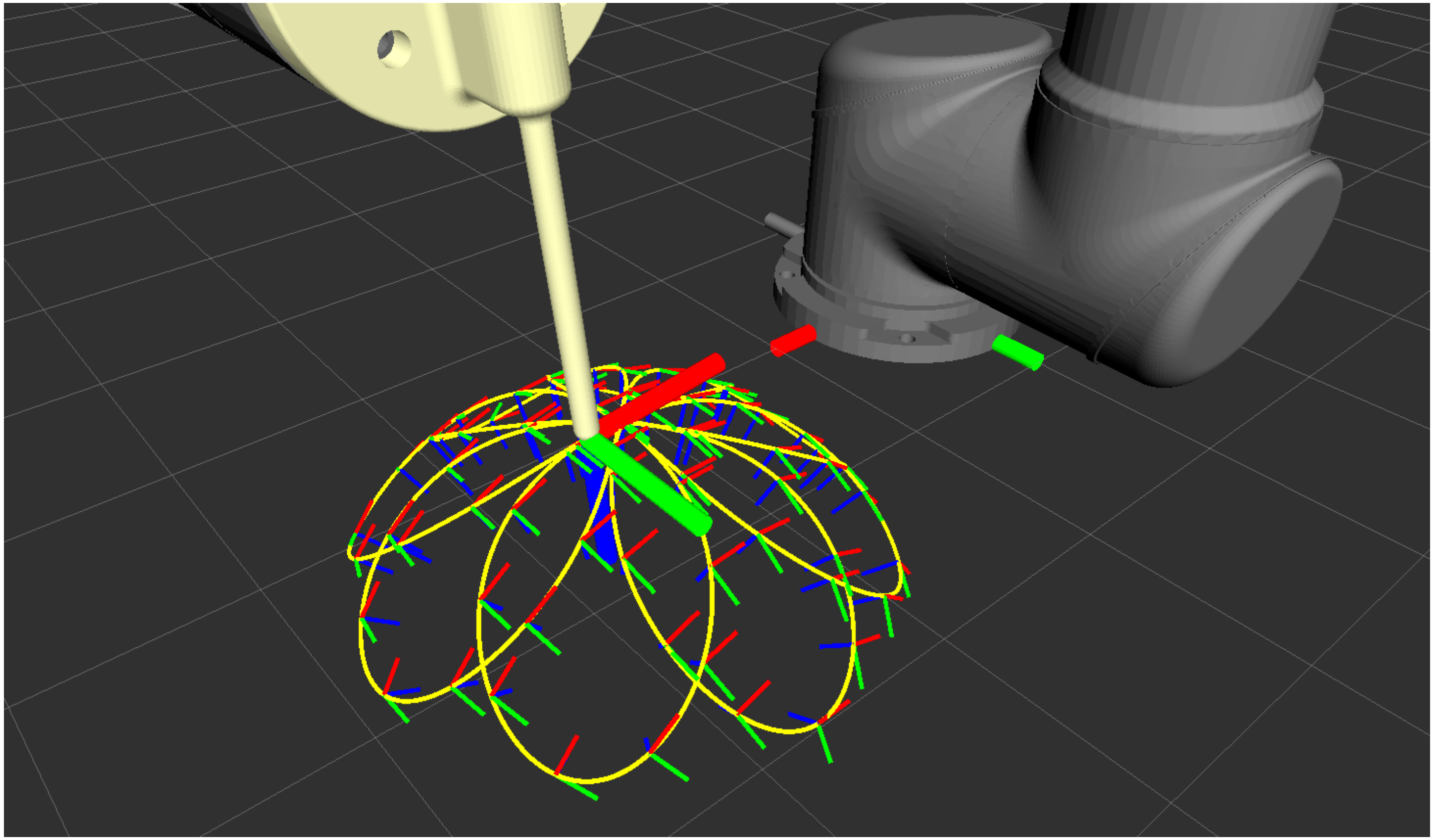
```
descartes_planner::DensePlanner planner;

descartes_planner::SparsePlanner planner;
```

Exercise 5.1:

Descartes Path Planning



DESCARTES IMPLEMENTATIONS

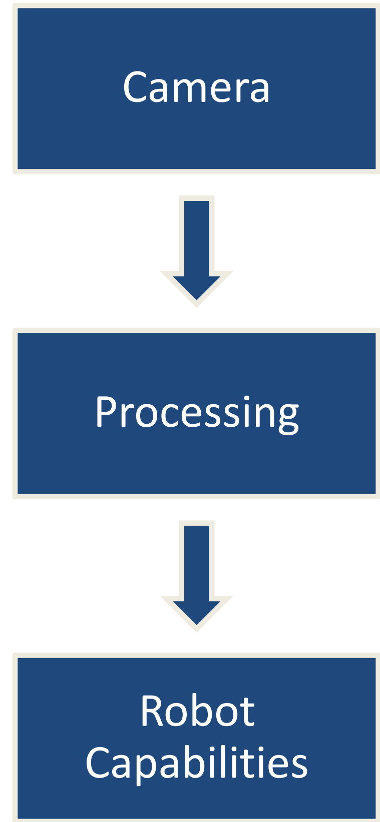


These points have a free degree of freedom, but they don't have to.

BUILDING A PERCEPTION PIPELINE

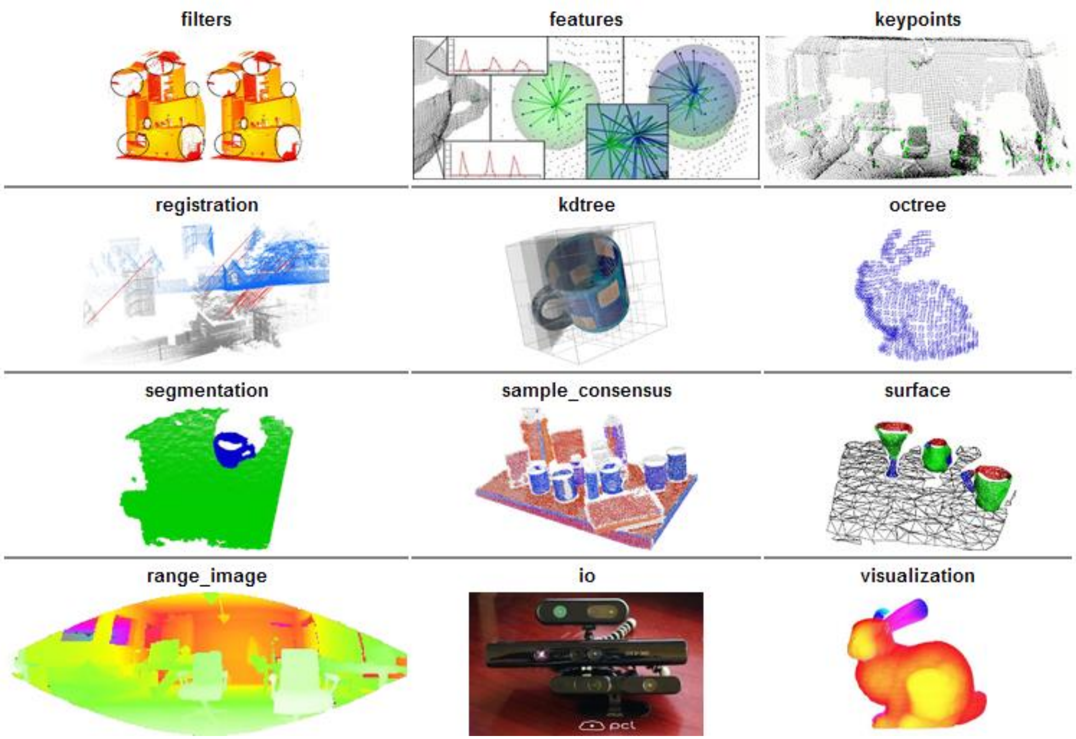
Perception Processing Pipeline

- Goal: Gain knowledge from sensor data
- Process data in order to
  - Improve data quality ➡ filter noise
  - Enhance succeeding processing steps ➡ reduce amount of data
  - Create a consistent environment model ➡ Combine data from different view points
  - Simplify detection problem ➡ segment interesting regions
  - Gain knowledge about environment ➡ classify surfaces



Perception Libraries (PCL)

- Point Cloud Library (PCL) - http://pointclouds.org/
  - Focused on 3D Range(Colorized) data



http://pointclouds.org

Perception Pipeline

Overall Process

1. 3D Camera
2. Processing
3. Robot Capabilities

Perception Process

1. Obtain PointCloud
2. Convert PointCloud ROS->PCL
3. Filter PointCloud
4. Convert PointCloud PCL->ROS
5. Publish PointCloud
6. Broadcast Transform*

PCL Methods

1. Voxel Grid
2. PassThrough
3. RANSAC Plane Segmentation
4. Others...

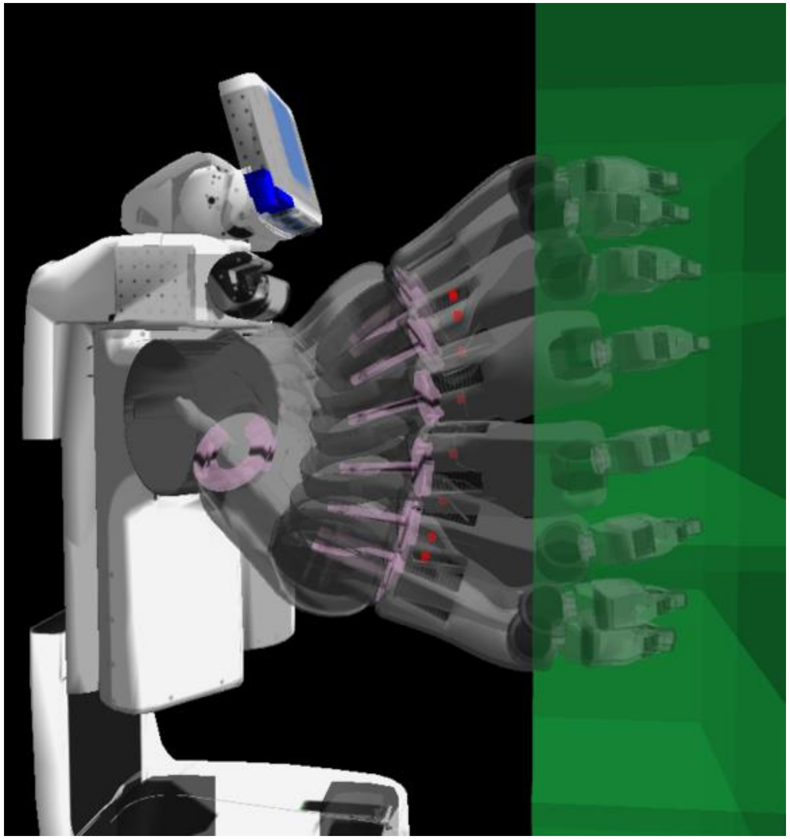
Exercise 5.1

- Exercise 5.1 - https://github.com/ros-industrial/industrial_training/wiki/Building-a-Perception-Pipeline

Introduction to STOMP

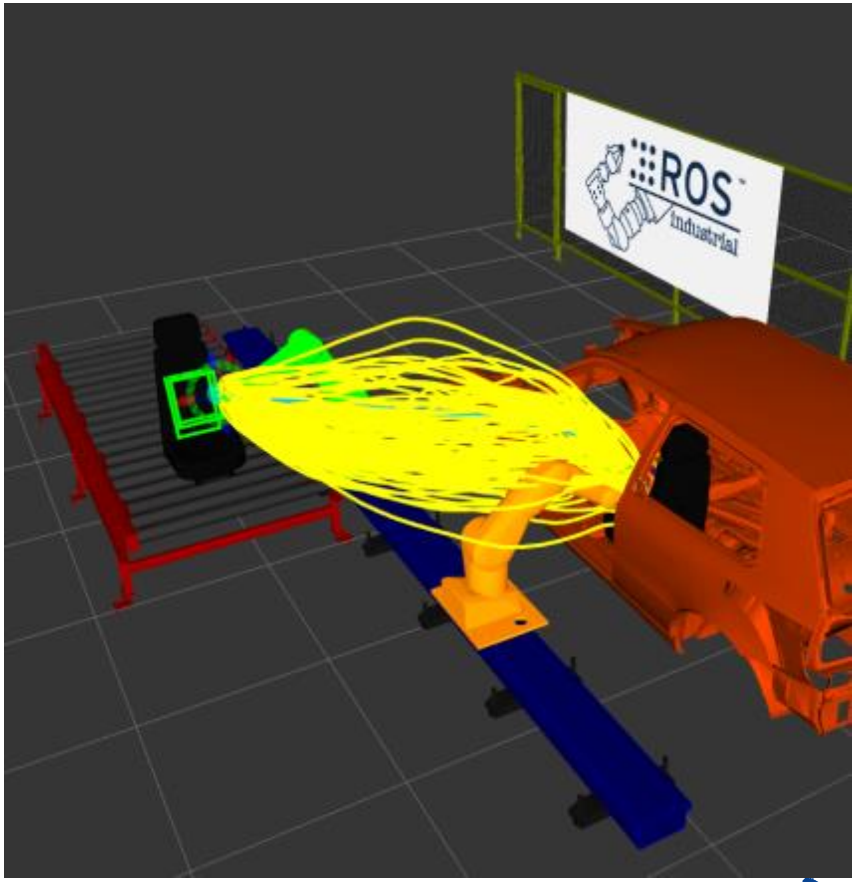
Introduction to STOMP

- Stochastic Trajectory Optimization Planner
- Optimization-based planner that generates smooth well behaved collision free motion paths in reasonable time.
- Original work by (Mrinal Kalakrishnan, Sachin Chitta, Evangelos Theodorou, Peter Pastor, Stefan Schaal, ICRA 2011)
- PI^2 (Policy Improvement with Path Integrals, Theodorou et al, 2010) algorithm
- The STOMP ROS package was first introduce in Hydro which was a direct implementation of the ICRA 2011 paper.

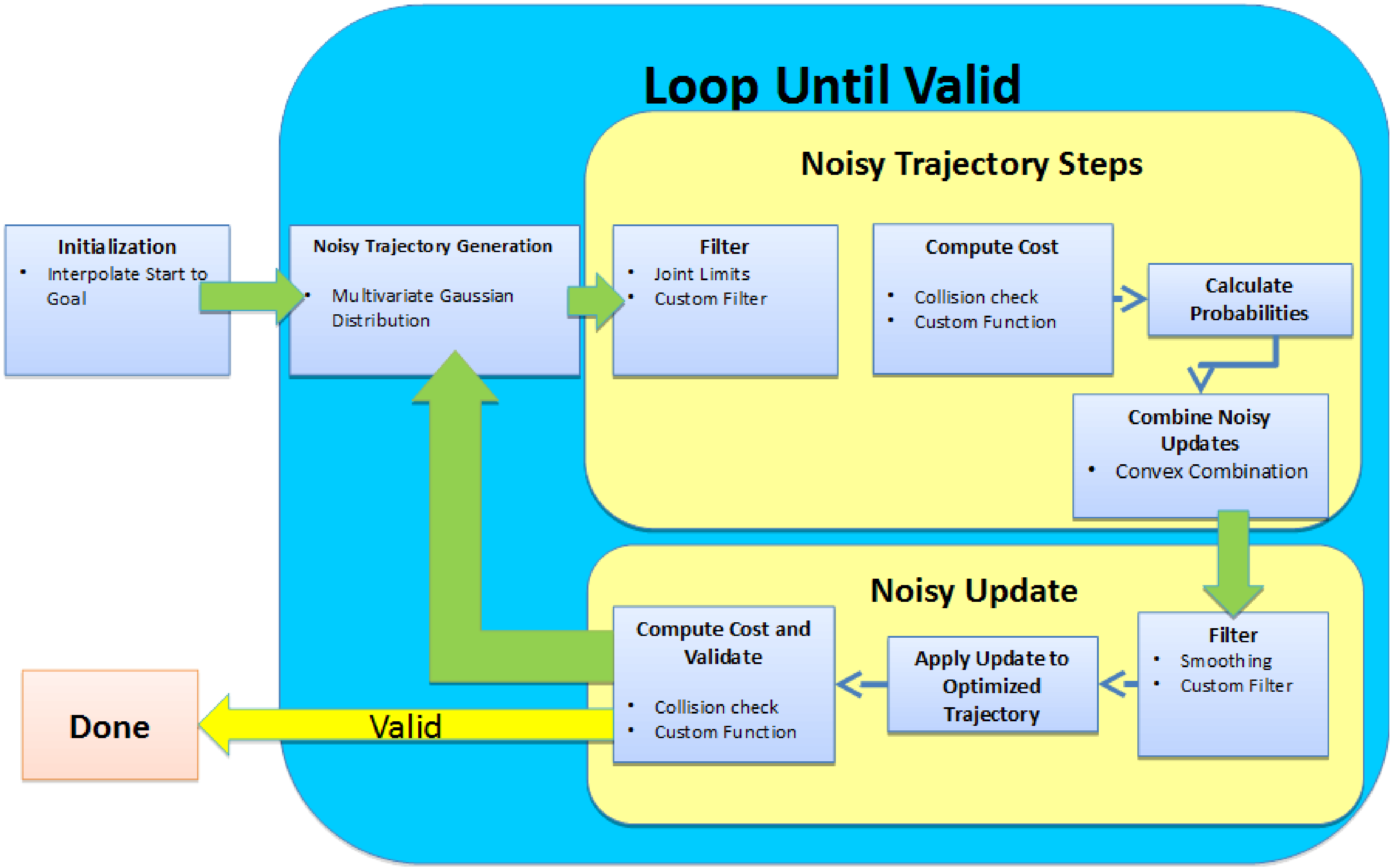


Introduction to STOMP

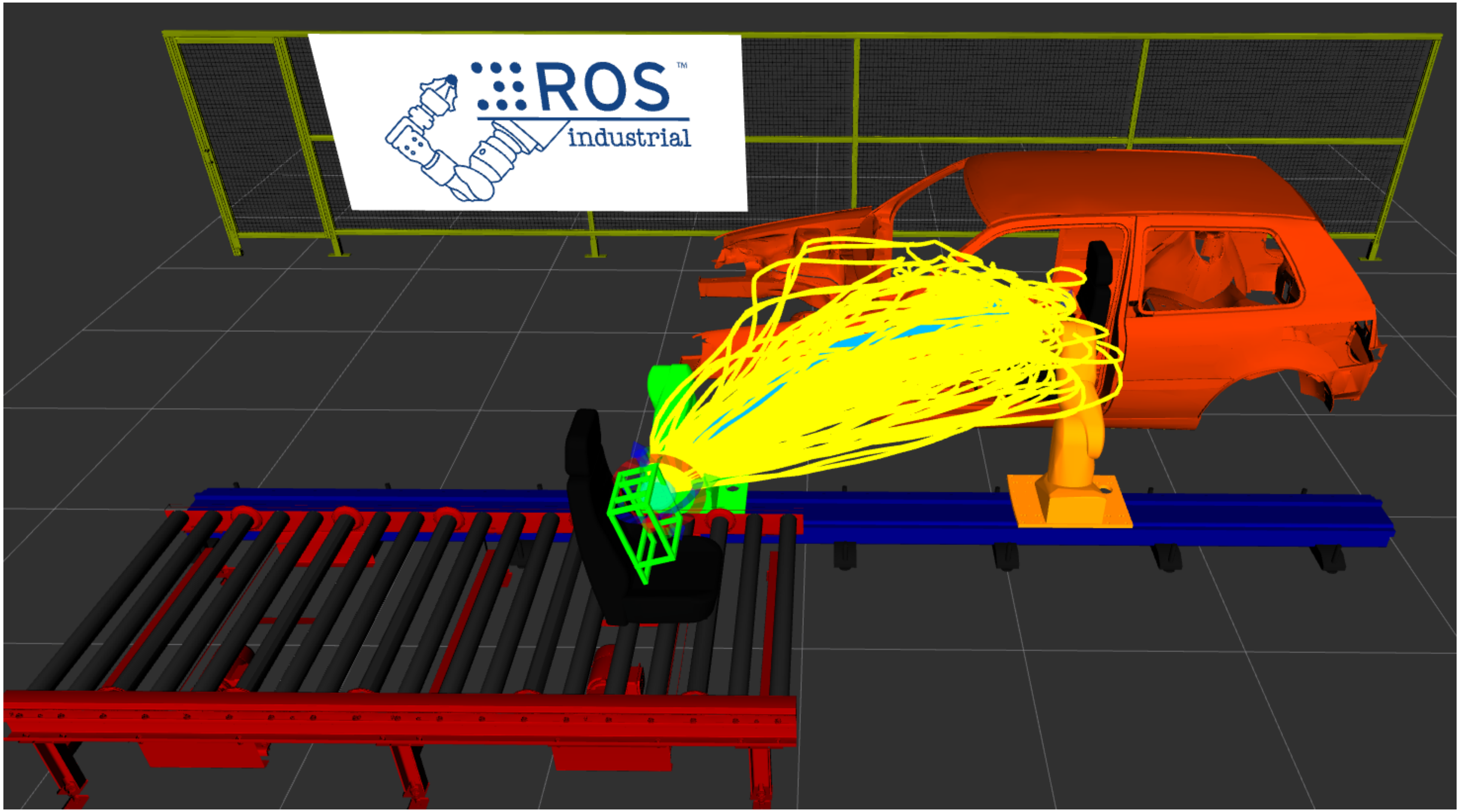
- Generates smooth well behaved motion plans in reasonable time.
- All steps of the algorithm are implemented through plugins and configurable via yaml file.
- Can Incorporates additional objective functions such as torque limits, energy and tool constraints.
- Cost functions that don’t need to be differentiable.
- Can use distance field and spherical approximations to quickly compute distance queries and collision costs.



Introduction to STOMP

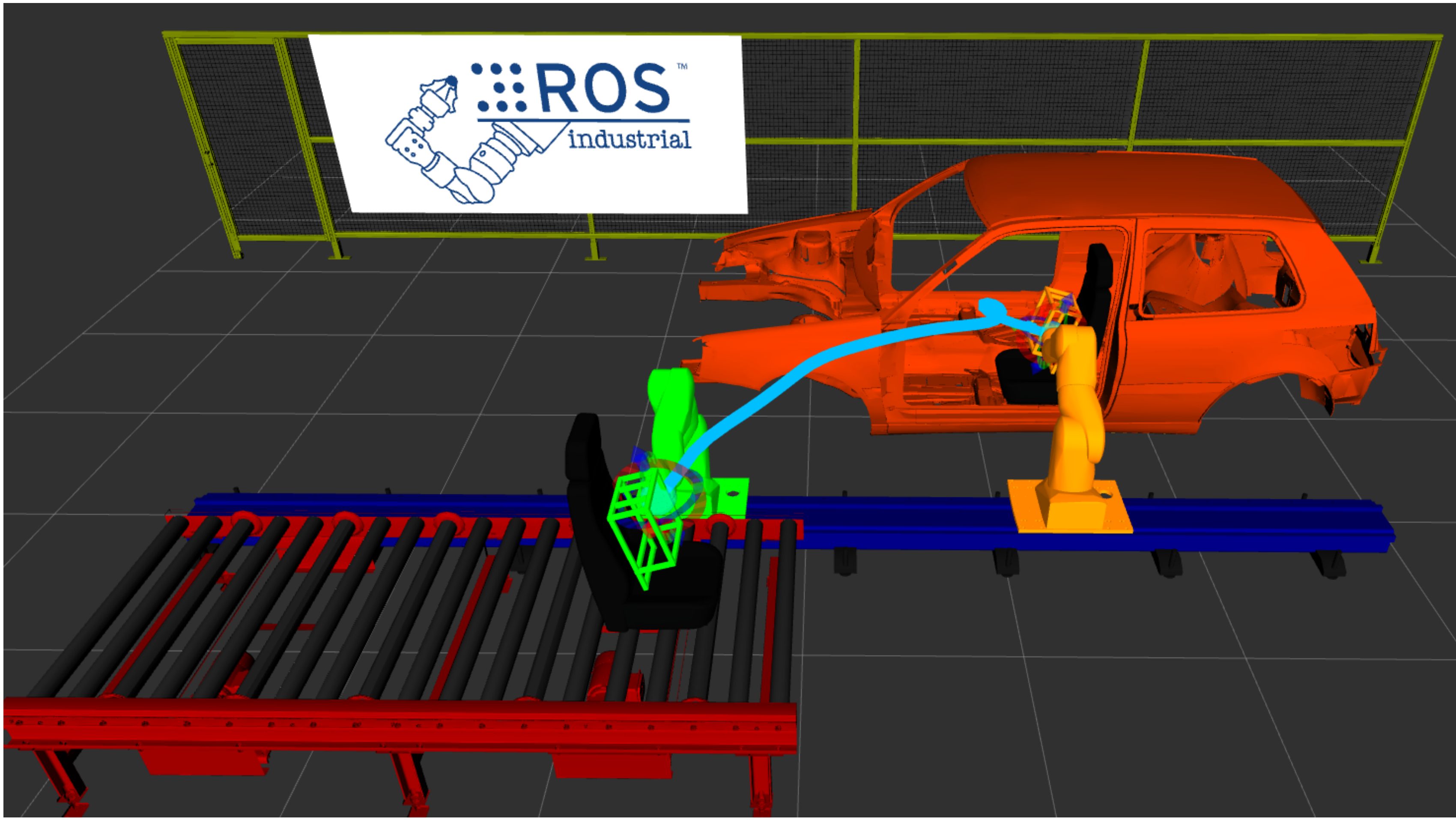


Introduction to STOMP



Noisy Generation

Introduction to STOMP



Noisy Update Smoothing

Exercise 5.2

Introduction to STOMP